University of Pittsburgh

Algorithms and Data Structures 2
CS 1501

Spring 2022

Sherif Khattab

ksm73@pitt.edu

(Slides are adapted from Dr. Ramirez's and Dr. Farnan's CS1501 slides.)

Announcements

- Upcoming deadlines:
  - Lab 11 due on 4/15
  - Homework 12 due on 4/18
  - Assignment 3 and 4 due on 4/18
  - Lab 12 due on 4/22
  - Bonus Lab due on 5/2
  - Assignment 5 due on 5/2
  - Bonus Homework due on 5/2

Previous lecture …

- Cryptography
    - Caesar Cipher
    - One-Time Pads
    - Symmetric Encryption
    - Asymmetric Encryption
        - RSA

RSA keypair example notes

- p and q must be prime
- $n = p * q$
- $\varphi(n) = (p - 1) * (q - 1)$
- Choose e such that
  - $1 < e < \varphi(n)$ and $GCD(e, \varphi(n)) = 1$
- Solve $XGCD(\varphi(n), e) = 1 = \varphi(n) * (-z) + e * d$
- Compute the ciphertext c as:
  - $c = m^e \pmod{n}$
- Recover m as:
  - $m = c^d \pmod{n}$

Implementation concerns

- Encryption/decryption:
  - How can we perform efficient exponentiations?
- Key generation:
  - We can do multiplication, XGCD for large integers
  - What about finding large prime numbers?

Exponentiation

- x^y
- Can easily compute with a simple algorithm:

```
ans = 1
i = y
while i > 0:
   ans = ans * x
   i--
```

- Runtime?

Just like with multiplication, let's consider large integers...

- Runtime = # of iterations * cost to multiply
- So how many iterations?
    - Single loop from 1 to y, so linear, right?
        - What is the size of our input?
            - n
                - The *bitlength* of y…
        - So, linear in the *value* of y…
            - But, increasing n by 1 doubles the number of iterations
    - $\Theta(2^n)$
        - Exponential in the *bitlength* of y

Runtime Analysis

y iterations

n-digit

$$y = \boxed{2^n - 1}$$

$x^{y-1} * x$: last iteration

$\Downarrow$?

n

x^2 : $2n$

x^4 : $4n$

x^{y-1} : $(y-1) * n = (2^n - 1 - 1) * n$

$= \theta(2^n * n)$

multiplication $(2^n * n)^{\cancel{2}1.58}$

iterations * multiplication runtime

$= y * (2^n * n)^{1.58}$

$= 2^n * (2^n * n)^{1.58}$

This is RIDICULOUSLY BAD

- Assuming 512 bit operands, 2^{512}:
  - 13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298166903427690031858186486050853753882811946569946433649006084096
- Assuming we can do mults in 1 cycle…
  - Which we *can't* as we learned last lecture
- And further that these operations are completely parallelizable
- 16 4GHz cores = 64,000,000,000 cycles/second
  - $(2^{512} / 64000000000) / (3600 * 24 * 365) =$
    - $6.64 * 10^{135}$ years to compute

This is way too long to do exponentiations!

- So how do we do better?
- Let’s try divide and conquer!
- $x^y = (x^{(y/2)})^2$
    - When y is even, $(x^{(y/2)})^2 * x$ when y is odd
- Analyzing a recursive approach:
    - Base case?
        - When y is 1, x^y is x; when y is 0, x^y is 1
    - Runtime?

Runtime Analysis

$$\log_2 y = \log_2 (2^n \cancel{-1})$$

$$= n$$

$$\Theta(\text{work at root})$$

$$= \Theta\left(\left(2^{n-1} \cdot n\right)^{1.58}\right)$$

But we need to do expensive mult in each call

- We need to do $\Theta((2^{(n-1)} * n)^2)$ work in just the root call!
    - Our runtime is dominated by multiplication time
        - Exponentiation quickly generates HUGE numbers
        - Time to multiply them quickly becomes impractical

Can we do better?

- We go "top-down" in the recursive approach
    - Start with y
    - Halve y until we reach the base case
    - Square base case result
    - Continue combining until we arrive at the solution
- What about a "bottom-up" approach?
    - Start with our base case
    - Operate on it until we reach a solution

A bottom-up approach

- To calculate x^y

```
ans = 1
foreach bit in y:
    ans = $ans^2$
    if bit == 1:
        ans = ans * x
```

From most to least significant

Bottom-up exponentiation example

- Consider x^y where y is 43 (computing x^{43})
- Iterate through the bits of y (43 in binary: 101011)
- ans = 1

$\text{ans} = 1^2 \qquad = 1$

$\text{ans} = 1 * x \qquad = x$

$\text{ans} = x^2 \qquad = x^2$

$\text{ans} = (x^2)^2 \qquad = x^4$

$\text{ans} = x^4 * x \qquad = x^5$

$\text{ans} = (x^5)^2 \qquad = x^{10}$

$\text{ans} = (x^{10})^2 \qquad = x^{20}$

$\text{ans} = x^{20} * x \qquad = x^{21}$

$\text{ans} = (x^{21})^2 \qquad = x^{42}$

$\text{ans} = x^{42} * x \qquad = x^{43}$

Bottom-up Exponentiation Example 1

x^y 23^{17}

$17 : 10001$

16

$ans = 1$

$ans = 1 * 1 = 1$

$ans = 1 * 23 = 23$

$ans = 23 * 23 = ? (23)^2$

$ans = ans * ans = (23)^4$

$ans = ans * ans = (23)^8$

$ans = ans * ans = (23)^{16}$

$ans = ans * 23 = (23)^{17}$

Bottom-up Exponentiation Example 2

$x^y = 5^{21}$

$$21 = \overset{16}{1}0\overset{4}{1}0\overset{1}{1}$$

$\uparrow$

$ans = 1$

$ans = ans * ans = 1 * 1$

$= ans * x = 1 * 5$

$ans = ans * ans = 25 = 5^2$

$ans = ans * ans = 5^4$

$ans = ans * x = 5^4 * 5 = 5^5$

$ans = ans * ans = 5^{10}$

$ans = ans * ans = 5^{20}$

$ans = ans * x = 5^{20} * 5 = 5^{21}$

Bottom-up Exponentiation

x^y:

```
ans = 1
for each bit b of y
    ans = (ans * ans) mod m
    if b == 1
        ans = (ans * x) mod m
```

n iterations

n^2

n^2

$m^e \bmod m$

$\Theta(n \cdot n^2)$

Does this solve our problem with mult times?

- Nope, still squaring ans everytime
    - We'll have to live with huge output sizes
- This does, however, save us recursive call overhead
    - Practical savings in runtime

Efficient exponentiation for RSA

Does this solve our problems?

```
ans = 1
foreach bit in y:
        ans = (ans$^2$ mod n)
        if bit == 1:
                ans = (ans*x mod n)
```

- How can we improve runtime for RSA exponentiations?
  - ○ Don't actually need x^y
    - ■ Just need (x^y mod n)

RSA Example 1

1) $p = 7$
   $q = 11$
2) $n = p * q = 77$
3) $\phi(n) = (p-1)(q-1) = 60$
4) $e = 13$
5) Use XGCD to compute $d = 37$
   $1 = y * 60 + d * 13$

$60 = 4 * 13 + 8 \Leftarrow 8 = 60 - 4 * 13$

$GCD(13,8)$

$13 = 1 * 8 + 5 \Leftarrow 5 = 13 - 1 * 8$

$GCD(8,5)$

$8 = 1 * 5 + 3 \Leftarrow 3 = 8 - 1 * 5$

$GCD(5,3)$

$5 = 1 * 3 + 2 \Leftarrow 2 = \underline{5 - 1 * 3}$

$GCD(3,2)$

$3 = 1 * 2 + 1 \Leftarrow 1 = 3 - 1 * 2$

$GCD(2,1) = 1$

$= 3 - 1 * (5 - 1 * 3)$

$= 2 * 3 - 5$

$= 2 * (8 - 1 * 5) - 5$

$= 2 * 8 - 3 * 5$

$= 2 * 8 - 3 * (13 - 1 * 8)$

$= 5 * 8 - 3 * 13$

$= 5 * (60 - 4 * 13) - 3 * 13$

$1 = 5 * 60 \underset{d}{\boxed{-23}} * 13$

$$\begin{array}{r} 300 \\ -299 \\ \hline 1 \end{array} \qquad \begin{array}{r} 230 \\ 69 \\ \hline 299 \end{array}$$

$d = -23 + 60 = 37$

RSA Example 2

$p = 5$

$q = 13$

$n = p * q = 65$

$\phi(n) = (p-1) * (q-1) = 4 * 12 = 48$

$1 < e < \phi(n) \quad \underline{GCD}(e, \phi(n)) = 1$

$e = 7$

$1 = x . \phi(n) + d . e$

$= \underline{x} . 48 + \underline{d} . 7$

$GCD(48, 7)$

$48 = 6 * 7 + 6 \leftarrow 6 = \underbrace{48 - 6 * 7}$

$GCD(7, 6)$

$7 = 1 * 6 + 1 \leftarrow 1 = 7 - 1 * 6$

$GCD(6, 1) \qquad = 7 - 1 * (48 - 6 * 7)$

$6 = 6 * 1 + 0$

$GCD(1, 0) = 1 \qquad 1 = \underset{d}{7} * 7 \underset{x}{-1} * 48$

$d = 7$

$c = m^e \bmod n$

$m \overset{?}{=} c^d \bmod n$

$\underline{m = 42}$

RSA Example 3

1) $p = 13 \quad q = 7$

2) $n = p.q = 13.7 = 91$

3) $\phi(n) = \phi(p).\phi(q) = \overset{12}{(p-1)}\overset{6}{(q-1)}$

$= 72$

└ Count of integers $1 \leq k < n$ s.t. $GCD(n,k) = 1$

4) Select $e = 11$

. $1 < e < \phi(n)$

. $GCD(e, \phi(n)) = 1$

5) Compute d using XGCD

$d = \frac{1}{e} \bmod \phi(n)$

$d = e^{-1} \bmod \phi(n)$

$ed = 1 \bmod \phi(n)$

$ed = z\,\phi(n) + 1$

$1 = ed + (-z)\phi(n)$

$GCD(e, \phi(n)) = e\overset{y}{d} + \overset{x}{(-z)}\phi(n)$

$XGCD(\phi(n), e) \rightarrow 1$

RSA Example 3 (contd.)

$GCD(72, 11) = GCD(11, 72 \% 11)$

$72 = 6 * 11 + 6 \Leftarrow 6 = 72 - 6 * 11$

$GCD(11, 6) = GCD(6, 11 \% 6)$

$11 = 1 * 6 + 5 \Leftarrow 5 = 11 - 1 * 6$

$GCD(6, 5) = GCD(5, 6 \% 5)$

$6 = 1 * 5 + 1 \Leftarrow$

$GCD(5, 1) = GCD(1, 5 \% 1)$

$= GCD(1, 0) = 1$

$1 = 6 - 1 * 5$

$= 6 - 1 * (11 - 1 * 6)$

$= 2 * 6 - 1 * 11$

$= 2 * (72 - 6 * 11) - 1 * 11$

$= \boxed{2} * 72 \boxed{-13} * 11$

$x \quad y = d$

$$\begin{array}{r} -143 \\ +144 \\ \hline 1 \end{array}$$

$d = -13 = -13 \bmod 72$

$= 59$

6) $m^e \bmod n$ $\quad 89^{11} \bmod 91 = 45$

7) ${}^d \bmod n$ $\quad 45^{59} \bmod 91 = 89$

RSA Example 4

1) $P = 17$ $q = 31$

2) $n = p \cdot q = 527$

$$\begin{array}{r} 310 \\ 217 \\ \hline 527 \end{array}$$

3) $\phi(n) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$
$= 16 \cdot 30 = 480$

└ Count of integers s.t. $GCD(k,n)=1$
$1 \leq k < n$

4) Select $e = 7$

- $1 < e < \phi(n)$
- $GCD(\phi(n), e) = 1$

5) Compute d using XGCD

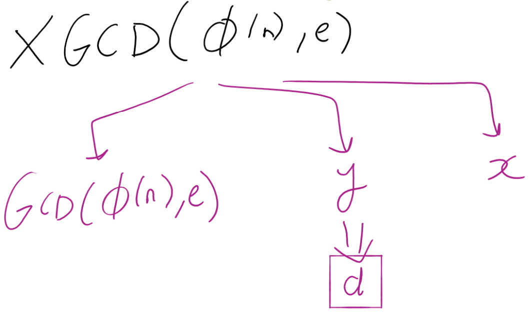
$d = \frac{1}{e} \bmod \phi(n)$

$ed = 1 \bmod \phi(n)$

$= z\,\phi(n) + 1$

$1 = ed + (-z)\,\phi(n)$

$GCD(\phi(n),e) = e\underbrace{d}_{y} + \underbrace{(-z)}_{x}\phi(n)$



RSA Example 4 (contd.)

$GCD(480, 7) = GCD(7, 480 \% 7)$

$480 = 68 * 7 + \underline{\underline{4}} \Rightarrow 4 = 480 - 68 * 7$

$GCD(7, 4) = GCD(4, 7 \% 4)$

$7 = 1 * 4 + 3 \Rightarrow 3 = 7 - 1 * 4$

$GCD(4, 3) = GCD(3, 4 \% 3)$

$4 = 1 * 3 + 1 \Leftarrow$

$GCD(3, 1) = GCD(1, 3 \% 1)$

$= GCD(1, 0) = 1$

$$1 = 4 - 1 * 3$$

$$= 4 - 1 * (7 - 1 * 4)$$

$$= 2 * 4 - 1 * 7$$

$$= 2 * (480 - 68 * 7) - 1 * 7$$

$$1 = \underset{x}{2} * 480 \underset{d}{-137} * 7$$

$$d = -137 \% \phi(n) = 480 - 137 = \boxed{d = 343}$$

$$m^e \bmod n = m^7 \bmod 527$$

$$106^7 \bmod 527 = 115$$

$$C^d \bmod n = 15^{343} \bmod 527 = 106$$

OK, but how does m^{ed} = m mod n?

- Feel free to look up the proof using Fermat’s little theorem
  - Knowing this proof is **NOT** required for the course
  - Knowing how to generate RSA keys and encrypt/decrypt **IS**
- For this course, we’ll settle with our example showing that it *does* work

Why is RSA secure?

- 4 avenues of attack on the math of RSA were identified in the original paper:
  - Factoring n to find p and q
  - Determining φ(n) without factoring n
  - Determining d without factoring n or learning φ(n)
  - Learning to take eth roots modulo n

Factoring n

- To the best of our knowledge, this is *hard*
  - A 768 bit RSA key was factored one time using the best currently known algorithm
    - Took 1500 CPU years
      - 2 years of real time on hundreds of computers
    - Hence, large keys are pretty safe
      - 2048 bit keys are a pretty good bet for now

What about determining φ(n) without factoring n?

- Would allow us to easily compute d because $ed = 1 \bmod \varphi(n)$
- Note:
  - $\varphi(n) = n - p - q + 1$
    - $\varphi(n) = n - (p + q) + 1$
    - $(p + q) = n + 1 - \varphi(n)$
  - $(p + q) - (p - q) = 2q$
  - Now we just need (p - q)…
    - $(p - q)^2 = p^2 - 2pq + q^2$
    - $(p - q)^2 = p^2 + 2pq + q^2 - 4pq$
    - $(p - q)^2 = (p + q)^2 - 4pq$
    - $(p - q)^2 = (p + q)^2 - 4n$
    - $(p - q) = \sqrt{((p + q)^2 - 4n)}$
- If we can figure out φ(n) efficiently, we could factor n efficiently!

$$\phi(n) = (p-1)(q-1)$$
$$= pq - q - p + 1$$
$$= n - q - p + 1$$

Reduction and RSA Security

Problem A reduces to Problem B

iff

a solution for A leads to a solution for B

RSA Attack ⇒ Integer Factorization

input: large integer n

output: prime factors

Believed to be hard

d ⇒ factor n efficiently

Determining d without factoring n or learning φ(n)?

- If we know, d, we can get a multiple of φ(n)
  - ed = 1 mod φ(n)
  - ed = kφ(n) + 1
    - For some k
  - ed - 1 = kφ(n)
- It has been shown that n can be efficiently factored using any multiple of φ(n)
  - Hence, this would provide another efficient solution to factoring!

Learning to take eth roots modulo n

- Conjecture was made in 1978 that breaking RSA would yield an efficient factoring algorithm
  - To date, it has been not been proven or disproven

This all leads to the following conclusion

- Odds are that breaking RSA efficiently implies that factoring can be done efficiently.
- Since factoring is probably hard, RSA is probably safe to use.

RSA Implementation concerns

- Encryption/decryption:
  - How can we perform efficient exponentiations?
- Key generation:
  - We can do multiplication, XGCD for large integers
  - What about finding large prime numbers?

Prime testing option 1: BRUTE FORCE

- Try all possible factors of x
    - 1 .. sqrt(x)
        - aka 1 .. sqrt($2^{size(x)}$)
            - For a total of $2^{(size(x)/2)}$ factor checks
- A factor check should take about the same amount of time as multiplication
    - $size(x)^2$

$$\sqrt{x} = \sqrt{2^n} = 2^{n/2}$$

- So our runtime is $\Theta(2^{(size(x)/2)} * size(x)^2)$

Option 2: A probabilistic approach

- Need a method test : $Z \times Z \rightarrow \{T, F\}$
  - If test(x, a) = F, x is composite based on the witness a
  - If test(x, a) = T, x is probably prime based on the witness a
- To test a number x for primality:
  - Randomly choose a witness a
    - if test(x, a) = F, x is composite
    - if test(x, a) = T, loop
- Possible implementations of test(x, a):
  - Miller-Rabin, Fermat's, Solovay–Strassen

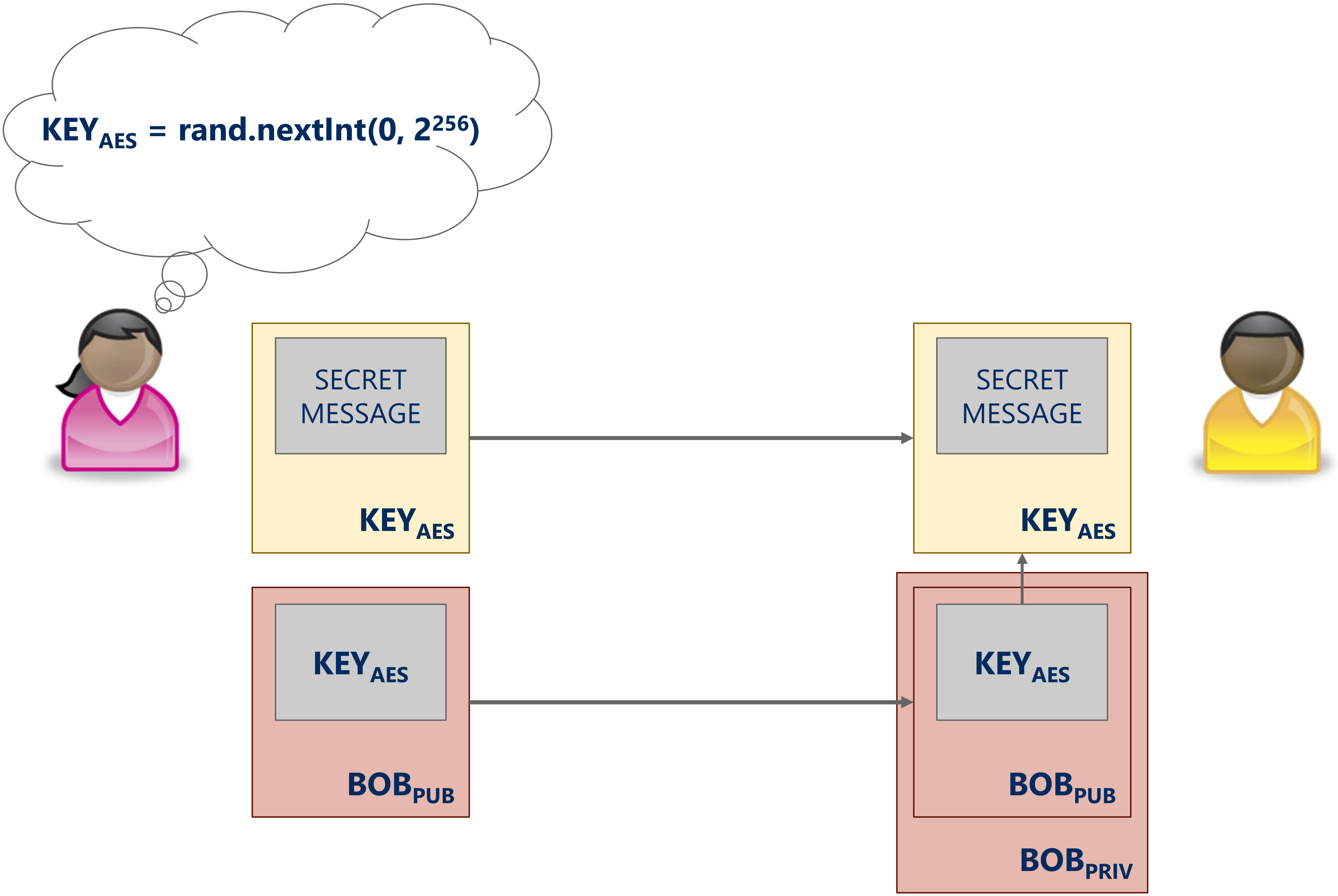
often probability ≈ 1/2

k repetitions leads to probability that x is composite ≈ $1/2^k$

RSA still slower (generally) than symmetric encryption

- If only we could have the speed of symmetric encryption without the key distribution woes!
  - What if we transmitted symmetric crypto keys with RSA?
    - RSA Envelopes!
- Going back to Alice and Bob
  - Alice generates a random AES key
  - Alice encrypts her message using AES with this key
  - Alice encrypts the key using Bob’s RSA public key
  - Alice sends the encrypted message and encrypted key to Bob
  - Bob decrypts the AES key using his RSA private key
  - Bob decrypts the message using the AES key

RSA Envelope example



KEY_{AES} = rand.nextInt(0, 2^{256})

SECRET MESSAGE

KEY_{AES}

SECRET MESSAGE

KEY_{AES}

KEY_{AES}

BOB_{PUB}

KEY_{AES}

BOB_{PUB}

BOB_{PRIV}

Another fun use of RSA...

- Notice that encrypting and decrypting are inverses
    - $m^{ed} = m^{de}$ (mod n)
- We can "decrypt" the message first with a private key
- Then recover the message by "encrypting" with a public key
- Note that anyone can recover the message
    - However, they know the message must have come from the owner of the private key
        - Using RSA this way creates a *digital signature*

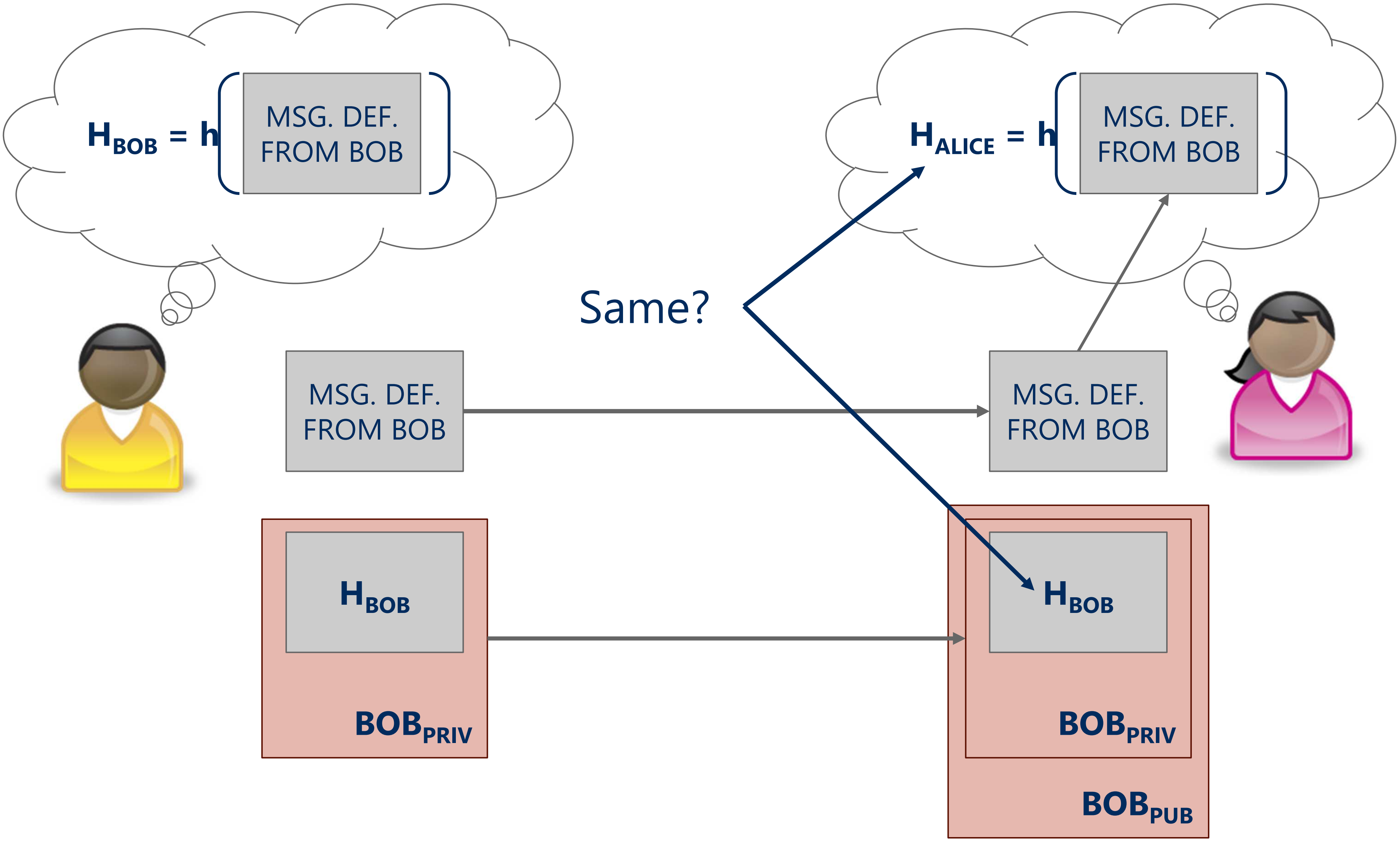
Do RSA signatures need to be slow?

- We encrypted symmetric crypto keys before to speed things up…
  - We'll need another crypto primitive to help out here
  - Cryptographically secure hash functions

Hashing for security (similarities)

- Cryptographically secure hash functions share properties with the hash functions we've already talked about in recitations:
    - Map from some input domain to a limited output range
        - Though output ranges are much larger here
            - For modern algorithms 224-512 bit output sizes
    - Time required to compute the hash is proportional to the size of the item being hashed
        - Though, practically, cryptographic hash functions are more expensive

Now just sign a hash of the message!



H_{BOB} = h(MSG. DEF. FROM BOB)

H_{ALICE} = h(MSG. DEF. FROM BOB)

Same?

MSG. DEF. FROM BOB

MSG. DEF. FROM BOB

H_{BOB}

BOB_{PRIV}

H_{BOB}

BOB_{PRIV}

BOB_{PUB}

Why does it work?

$$\left(H_{BoB}^{d} \bmod n\right)^{e} = H_{BoB}^{de} \bmod n$$

$$= H_{BoB}^{ed} \bmod n$$

$$= H_{BoB} \bmod n$$

What about collisions?

- If Bob signs a hash of the message “I’ll see you at 7”
- It could appear that Bob signed any message whose hash collides with “I’ll see you at 7”…
- If h(“I’ll see you at 7”) == h(“I’ll see you after I rob the bank”), Bob could be in alot of trouble
- An attack like this helped the Flame malware to spread
- This is also the reason Google is aiming to deprecate SHA-1

Hashing for security (differences)

- This is why cryptographically secure hash functions must support additional properties:
    - It should be infeasible to find two different messages with the same hash value
    - It should be infeasible to recover a message from its hash
        - Should require a brute force approach
    - Small changes to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value

Public key isn’t perfect

What do you do when a private key is compromised?

Final note about crypto

NEVER IMPLEMENT YOUR OWN CRYPTO

Use a trusted and tested library.

Master Method

- How can we analyze the runtime of a recursive algorithm?
    - Master Method!

So what's the runtime???

- Recursion really complicates our analysis…
- We'll use a *recurrence relation* to analyze the recursive runtime
  - Goal is to determine:
    - How much work is done in the current recursive call?
    - How much work is passed on to future recursive calls?
    - All in terms of input size

Recurrence relation for divide and conquer multiplication

- Assuming we cut integers exactly in half at each call
    - I.e., input bit lengths are a power of 2
- Work in the current call:
    - Shifts and additions are $\Theta(n)$
- Work left to future calls:
    - 4 more multiplications on half of the input size

- $T(n) = 4T(n/2) + \Theta(n)$

Soooo… what’s the runtime?

- Need to solve the recurrence relation
    - Remove the recursive component and express it purely in terms of n
        - A “cookbook” approach to solving recurrence relations:
            - The master theorem

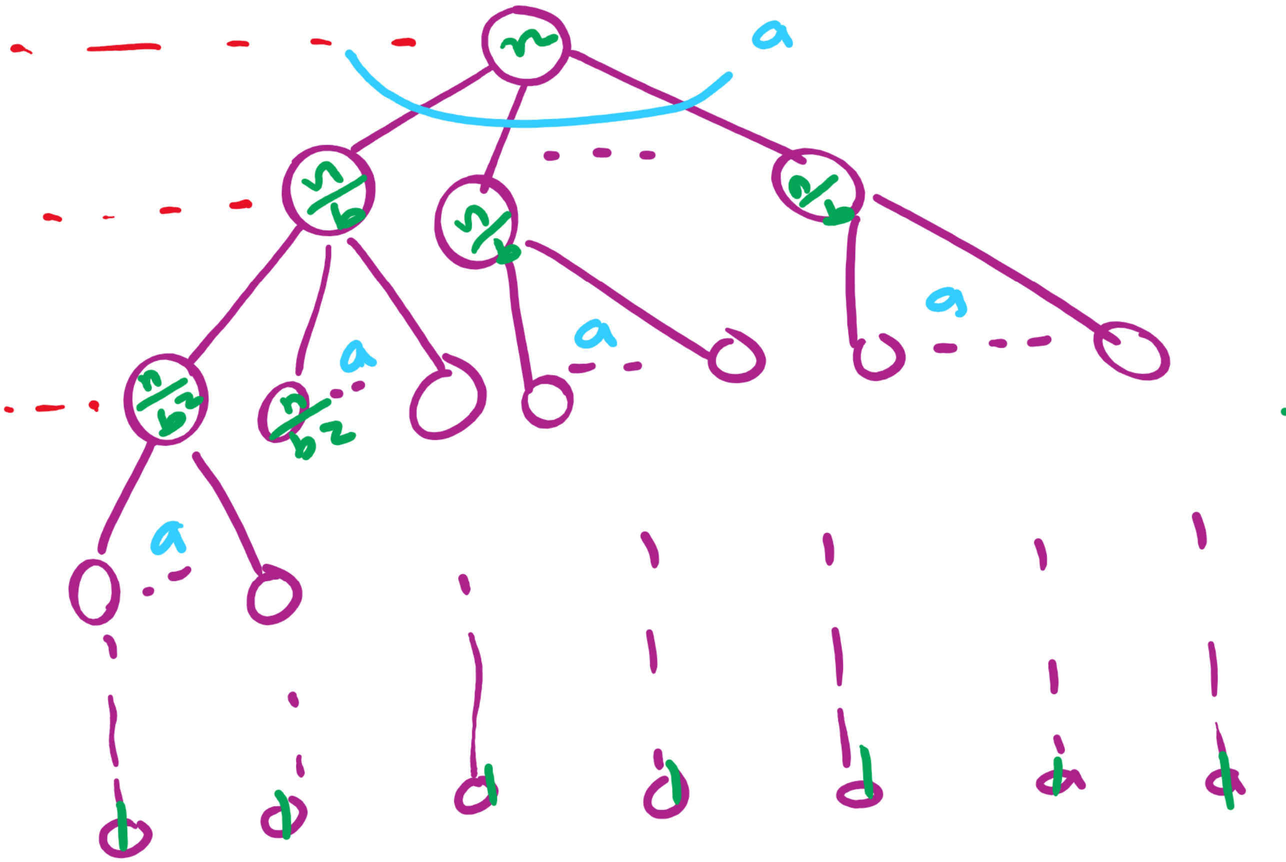
The master theorem

- Usable on recurrence relations of the following form:

$$T(n) = aT(n/b) + f(n)$$

- Where:
    - a is a constant >= 1
    - b is a constant > 1
    - and f(n) is an asymptotically positive function

Recursion Tree



#nodes

*

work/node

Please submit your reflections by using the CourseMIRROR App

If you are having a problem with CourseMIRROR, please send an email to coursemirror.development@gmail.com

8/29/2022